View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by PORTO@iris (Publications Open Repository TOrino - Politecnico di Torino)

POLITECNICO DI TORINO
Repository ISTITUZIONALE

Using Automatic Static Analysis to Identify Technical Debt

Original
Using Automatic Static Analysis to Identify Technical Debt / Antonio Vetro'. - STAMPA. - (2012), pp. 1613-1615. ((Intervento presentato al convegno 34th International Conference on Software Engineering (ICSE) tenutosi a Zurich, Switzerland nel June 2–9, 2012.

Availability:
This version is available at: 11583/2497506 since:

Publisher:
IEEE COMPUTER SOC

Published
DOI:10.1109/ICSE.2012.6227226

Terms of use:
openAccess

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

04 August 2020

Using Automatic Static Analysis to Identify Technical Debt

Antonio Vetrò
Politecnico di Torino/Fraunhofer CESE
Torino (Italy)/College Park, MD (USA)
antonio.vetro@polito.it

***Abstract*—The technical debt (TD) metaphor describes a tradeoff between short-term and long-term goals in software development. Developers, in such situations, accept compromises in one dimension (e.g. maintainability) to meet an urgent demand in another dimension (e.g. delivering a release on time). Since TD produces interests in terms of time spent to correct the code and accomplish quality goals, accumulation of TD in software systems is dangerous because it could lead to more difficult and expensive maintenance. The research presented in this paper is focused on the usage of automatic static analysis to identify Technical Debt at code level with respect to different quality dimensions. The methodological approach is that of Empirical Software Engineering and both past and current achieved results are presented, focusing on functionality, efficiency and maintainability.**

Keywords: **Technical Debt; Automatic Static Analysis; Software Maintenance; Software Quality Monitoring.**

I. Research Problem and Motivation

Systems evolve continuously to meet the changes of the surrounding technical and human context. Therefore a system needs maintenance, which according to Godfrey and German [1] can be corrective, adaptive, perfective or preventive.

Maintenance is a costly activity and recent studies have reported that the relative cost for maintaining software and managing its evolution now represents more than 90% of its total cost [2]. Due to time constraints and limited budget, maintenance is often done following a compromise between a well done change (e.g., preserving architectural design, employing good programming practices and standards, updating the documentation and testing thoroughly) and a change that simply works, done as quickly as possible and with as few resources as possible. The gap between the two types of change is the Technical Debt (TD).

The term "technical debt" was first coined by Ward Cunningham in [3], in which he presented the metaphor of "going into debt" every time a new release of a system is shipped. Cunningham explained that a little debt can speed up the software development in the short run and lower the cost of the current release, but every extra minute spent on not-quite-right code counts as interest on that debt in future releases [3]. In other words, the technical debt (TD) metaphor describes a tradeoff between short-term and long-term goals in software development, such as situation where developers accept compromises in one dimension (e.g. maintainability) to meet an urgent demand in another dimension (e.g. delivering a release on time).

TD produces interests that must be repaid to reduce it and restore the health of the system, avoiding that future changes in the system become too costly and hard to perform. Identifying TD, quantifying the values of debt and make proper decision making are some of the open issues in the current research on TD [4].

The research presented in this paper is focused on the problem of TD identification at code level.

II. Background and Related Work

Some techniques identify TD through source code analysis: code smells [5], automatic static analysis issues [6], grime build up [7] and modularity violations [8]) are some examples and have been partly evaluated to be valid TD indicators. The research presented herein is focused on automatic static analysis.

Source code analysis is a specific technique of reverse engineering [15] that consists in extracting information about a program from its source or artifacts (e.g., from Java byte code or execution traces) generated from the source code using automatic tools [16].

Automatic static analysis (ASA) tools analyze source or compiled code looking for violations of recommended programming practices ("issues") that might cause faults or might degrade some dimensions of software quality (e.g., maintainability, efficiency). Issues should be removed through refactoring to avoid future problems.

Some ASA issues are identified in the literature as good/bad defect detectors, both in industry [11] [13] [14] and open source software [12]. The overall finding is that not all ASA issues are related to defects in the software and the remaining ASA issues are related to other quality dimensions. However, no researches have differentiated the different quality dimensions degradations identified by ASA.

III. Approach and Uniqueness

TD derives from the compromise between desirable system properties and economic properties (e.g. effort, cost, time-to-market). Examples of desirable properties in the system are specific quality dimensions such as functionality, performance, reliability, maintainability, extendibility, usability, etc. The ISO\IEC standard 25010 [18] (that revises the ISO\IEC 9126 [19]) defines a quality model for the software product and specifies different quality characteristics: Functionality-Suitability, Reliability, Operability, Performance-Efficiency, Security, Compatibility, Maintainability, and Transferability.

The objective and uniqueness of my research is to differentiate the impact of using ASA tools on code quality with respect to the different quality dimensions. Knowing which ASA issues impact the different quality characteristics is an important step in better identify TD in code, and will enable both programmers and managers to prioritize

maintainability activities with respect to the quality dimensions of interest.

The idea is that quality dimensions of higher interest produce higher interests in the future because a lower quality in that area could compromise the software mission. However the quantification of the interests is currently beyond the scope of this research.

The methodological approach I adopt is that of empirical software engineering [20] [21], performing experiments and case studies. The research is part of my PhD plan, which was presented at the 5th International Doctoral Symposium on Empirical Software Engineering [22]. Results summarized in this paper were collected through experiments and case studies conducted both at Politecnico di Torino and at the Fraunhofer Center for Experimental Software Engineering - USA. I acknowledge all my supervisors and collaborators for their support.

IV. RESULTS

Results achieved so far are specific to three quality characteristics: Functionality-Usability, Performance Efficiency, and Maintainability. The choice of the quality characteristics to analyze first was driven by the availability and needs of industrial and academic partners.

A. Functionality-Suitability (Methodology: Case Studies)

We analyzed in two studies [9][10] the issues detected by FindBugs [6] on two pools of similar small programs (85 and 301 programs respectively), each of them developed by a different student, in order to verify which FindBugs issues were related to real defects in the source code. By analyzing the changes and test failures in both studies, we observed that a small percentage of issues (about 3%) were related to known defects in the code.

We also conducted another study (not yet published) with the Resharper tool analyzing the capability of the issues categories to identify the 80% most defect prone files and components of an industrial web application. We differentiated defects by ISO\IEC 9126 categories.

We found that only certain categories of Resharper issues were good indicators of faulty files and components, and those different categories of issues pointed to different quality dimensions according to the ISO\IEC 9126. The issues of these categories pointed to problems that can be associated to difficulty in the design of the code or a limited knowledge of the possibilities offered by the C# language. However, the Resharper issues were more efficient in identifying the 80% of defects and defect fixes rather than indicators of size and complexity.

B. Performance-Efficiency (Methodology: Experiments)

We conducted another study [17] with FindBugs but focused in efficiency, on which we empirically proved that refactoring code on the basis of certain ASA issues will improve its execution time.

We selected 20 issues and for each of them we set up two source code fragments: one containing the issue and the corresponding refactored version, functionally identical but without the issue. We set up three different platforms, isolated from network and other user programs, and then we executed the code fragments measuring the execution time of both code versions. We found that eleven issues have an actual negative impact on performance in all platforms (up to 6 times slower).

Moreover, in an industrial experiment not yet published, we quantitatively assessed the impact on time efficiency of three code patterns detectable by ASA: dead store to local variable, useless try-catch block and inefficient construction of Boolean objects. We refactored an industrial Java web application removing the three code patterns and we observed that the refactored software was about two fold faster (100ms less) than the original application.

C. Maintainability (Methodology: Case Study)

In yet another study not published, we compared the four TD techniques listed in section II (code smells, automatic static analysis issues, grime buildup, and modularity violations) and applied them to 13 versions of the Apache Hadoop open source software project. We collected and aggregated statistical measures to investigate whether the different techniques identified TD indicators in the same classes and whether those classes exposed high defect and change proneness. The latter metric is a proxy for Maintainability.

Although ASA issues were not directly associated with Maintainability issues, those with higher priority were associated with classes with intensive coupling. A possible explanation for this relation is that both indicators point, more than any others, to generally poorly designed code.

Moreover, the study demonstrated that the four approaches for TD identification have very little overlap and are therefore pointing to different problems in source code.

V. CONTRIBUTIONS

Assessing the impact of ASA on the different quality dimensions will enable developers and managers to better manage TD, detecting anomalies in the system that negatively impact specific qualities of interest. Therefore, maintenance activities could be prioritized according to the most degraded quality dimensions or those that deeply impact the software mission.

REFERENCES

[1] Godfrey, M.W.; German, D.M.; , "The past, present, and future of software evolution," *Frontiers of Software Maintenance, 2008. FoSM 2008.* , vol., no., pp.129-138, Sept. 28 2008-Oct. 4 2008

[2] Seacord, R., Plakosh, D. & Lewis, G. (2003). "Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices" (SEI Series in Software Engineering). Addison-Wesley

[3] Cunningham W. (1992). The WyCash Portfolio Management System. Addendum to the proceedings on Object-oriented programming systems, languages, and applications, pp. 29-30, 1992

[4] Brown N., Cai Y., Guo Y., Kazman R., Kim M., Kruchten P., Lim E., MacCormack A., Nord R., Ozkaya I., Sangwan R., Seaman C., Sullivan K., and Zazworka N. (2010). Managing technical debt in software-reliant systems. FoSER 2010: 47-52

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code, 1st ed. Addison-Wesley Professional, Jul. 1999.

[6] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” SIGPLAN Not., vol. 39, pp. 92–106, December 2004.

[7] C. Izurieta and J. Bieman, “How software designs decay: A pilot study of pattern evolution,” in Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on, sept. 2007, pp. 449 –451.

[8] S. Wong, Y. Cai, M. Kim, and M. Dalton, “Detecting software modularity violations,” in Proc. 33th International Conference on Software Engineering, May 2011, pp. 411–420.

[9] A. Vetro’, M. Torchiano, and M. Morisio, “Assessing the precision of findbugs by mining java projects developed at a university,” in Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, may 2010, pp. 110 –113.

[10] A. Vetro’, M. Morisio, and M. Torchiano, “An empirical validation of findbugs issues related to defects,” in To appear in Evaluation and Assessment in Software Engineering (EASE), EASE 2011, April 2011.

[11] N. Ayewah and W. Pugh, “The google findbugs fixit,” in Proceedings of the 19th international symposium on Software testing and analysis, ser. ISSTA ’10. New York, NY, USA: ACM, 2010, pp. 241–252.

[12] S. Kim and M. D. Ernst, “Prioritizing warning categories by analyzing software history,” in Proceedings of the Fourth International Workshop on Mining Software Repositories, ser. MSR ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 27–.

[13] C. Boogerd and L. Moonen, “Evaluating the relation between coding standard violations and faultswithin and across software versions,” in Mining Software Repositories, 2009. MSR ’09. 6th IEEE International Working Conference on, may 2009, pp. 41 –50.

[14] S. Kim and M. D. Ernst, “Which warnings should i fix first?” in Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ser. ESEC-FSE ’07. New York, NY, USA: ACM, 2007, pp. 45–54.

[15] Tonella P., Torchiano M., Du Bois B., Systa T. Empirical Studies in Reverse Engineering: State of the Art and Future Trends. EMPIRICAL SOFTWARE ENGINEERING, Vol. 12(5), pp. 551-571

[16] D. Binkley, “Source code analysis: A road map,” in Future of Software Engineering, 2007. FOSE ’07, may 2007, pp. 104 –119.

[17] A. Vetro’, M. Torchiano, and M. Morisio, “Quantitative assessment of the impact of automatic static analysis issues on time efficiency,” Inf-Q 2011, June 2011.

[18] ISO/IEC, ISO/IEC 9126. Software engineering – Product quality. ISO/IEC, 2001.

[19] ISO/IEC, ISO/IEC 25010. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. ISO/IEC, 2011.

[20] C. Wohlin, *Experimentation in software engineering: an introduction*, ser. Kluwer international series in software engineering. Kluwer Academic, 2000.

[21] Forrest Shull, Janice Singer, Dag I. K. Sjøberg , *Guide to advanced empirical software engineering*, Springer, 2007

[22] A. Vetro’, Empirical Assessment of the Impact of Automatic Static Analysis on Code Quality , In: IDOESE 2010 - 5th International Doctoral Symposium on Empirical Software Engineering, Bolzano, Italy, 15 September 2010